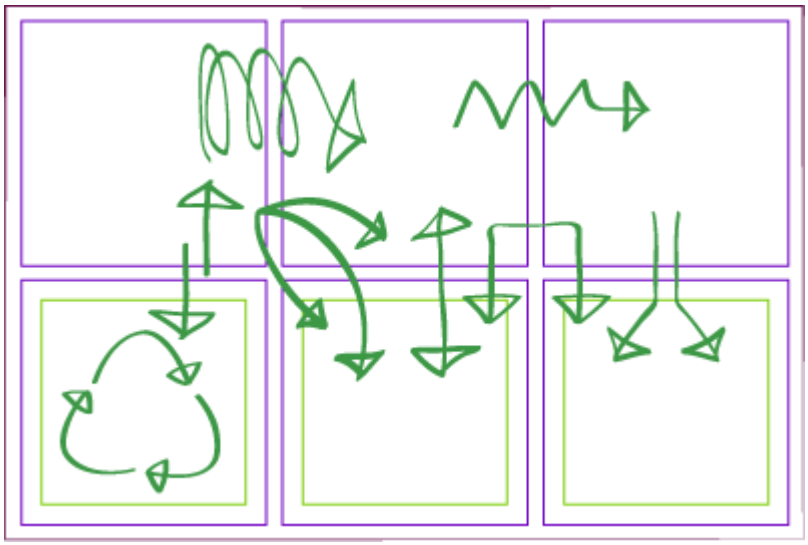
Manage state of data using ngRx

By – Piyush Nandan

Why ngRx?

If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update. At some point, you no longer understand what happens in your app as you have lost control over the when, why, and how of its state. When a system is opaque and non-deterministic, it’s hard to reproduce bugs or add new features.



By using ngrx/store you can actually get around this problem because you'll get a clear data flow in your app.

Since ngrx is highly inspired by redux, I would say that the same main principles apply:

Single source of truth

State is read-only

Changes are made with pure functions

So, in my opinion the biggest benefit is that you are able to easily track user interaction and reason about state changes because you dispatch actions and those lead always to one spot whereas with plain models you have to find all refernces and see what changes what and when.

Using ngrx/store also enables you to use devtools to see debug your state container and revert changes. Time travelling, I guess, was one of the main reasons for redux and that is pretty hard if you are using plain old models.

Getting Started

@ngrx/store is RxJS powered state management that is inspired by Redux. In NgRx, state is composed of a map of action reducer functions. Reducer functions are called with dispatched action and current or initial state and finally reducer returns immutable state.

Action: Action is state changes. It describes the fact that something happened but does not specify how the applications state changes.

StoreModule: StoreModule is a module in @ngrx/store API that is used to configure reducers in application module.

Store: It provides Store.select() and Store.dispatch() to work with reducers. Store.select() selects a selector and Store.dispatch() dispatches the type of action to reducer.

Technologies Used

Find the technologies being used in our example.

1. Angular 5.0.0

2. Angular CLI 1.5.0

3. NgRx/Store 4.1.1

4. TypeScript 2.4.2

5. Node.js 6.11.0

6. NPM 3.10.10

Install Angular CLI and NgRx/Store

Find the steps to install Angular CLI.

1. Make sure that Node and NPM are installed with minimum version as Node 6.9.x and NPM 3.x.x respectively.

2. Now run command using command prompt.

npm install -g @angular/cli This will install Angular CLI globally.

3. To generate a new project, run the command.

ng new my-app. Now install @ngrx/store. Go to the directory my-app using command prompt and run the command.

npm i @ngrx/store --save Now we are ready to work with NgRx/Store and Angular.

Create State

State is a single immutable data structure. We will create state as following.

```
export interface AppState {
   readonly tutorial: Tutorial[];
}
```

Create models

```
export interface Tutorial {
   name: string;
   url: string;
}
```

Create Action Classes

NgRx Action describes state changes. For each and every action, we need to create a class implementing Action and define type and payload where payload is optional.

```
export const ADD_TUTORIAL = '[TUTORIAL] Add';
export const REMOVE_TUTORIAL = '[TUTORIAL] Remove';

export class AddTutorial implements Action {
   readonly type = ADD_TUTORIAL;
   constructor(public payload: Tutorial) {}
}

export class RemoveTutorial implements Action {
   readonly type = REMOVE_TUTORIAL;
   constructor(public payload: number) {}
```

```
}

export type Actions = AddTutorial | RemoveTutorial;
```

Create Reducer

Reducer describes how the application state changes for any action. We will create reducer as following.

```
const initialState: Tutorial = {
   name: 'Initial',
   url: 'http://google.com'
};

export function reducer(state: Tutorial[] = [initialState], action:
TutorialActions.Actions ) {
   switch (action.type) {
      case TutorialActions.ADD_TUTORIAL:
         return [...state, action.payload];
      case TutorialActions.REMOVE_TUTORIAL:
         state.splice(action.payload, 1);
         return state;
      default:
         return state;
   }
```

}

Using StoreModule

StoreModule is a module in @ngrx/store API that is used to configure reducers in application module

```
@NgModule({
  imports: [  StoreModule.forRoot({ tutorial: reducer   }) ]})
```

Using Store.select() and Store.dispatch()

Store.select() and Store.dispatch() work with reducers to use them. Store.select() selects a selector and Store.dispatch() dispatches the type of action to reducer.

To use Store, create a property of Observable type in component.

```
tutorials: Observable<Tutorial[]>;
```

Now use dependency injection to instantiate Store and select the selector.

```
constructor(private store: Store<AppState>) {
    this.tutorials = store.select('tutorial');
  }
```

Now dispatch the action to change the state by reducer. Suppose to delete a particular name,

```
delTutorial(index) {
  this.store.dispatch(new TutorialActions.RemoveTutorial(index));}
```

Project Structure

```
my-app
|
|--src
|   |
|   |--app
|   |   |
|   |   |--actions
|   |   |   |
|   |   |   |--tutorial.action.ts
|   |   |
|   |   |--read (component)
|   |   |   |
|   |   |   |--read.component.html
|   |   |   |--read.component.ts
|   |   |
|   |   |--create (component)
|   |   |  |
|   |   |  |-create.component.html
|   |   |  |-create.component.ts
|   |   |  |
|   |   |--models
|   |   |   |
```

```
|   |   |    |--tutorial.model.ts
|   |   |
|   |   |--reducers
|   |   |   |
|   |   |   |--article.reducer.ts
|   |   |
|   |   |--app.component.ts
|   |   |--app.module.ts
|   |   |--app.state.ts
|   |
|   |--main.ts
|   |--index.html
|   |--styles.css
|
|--node_modules
|--package.json
```

Here , component interaction is happening between read and create component.

Create.component.html

```
<div class="row">
<div class="left col-md-6">
  <input type="text" placeholder="name" #name>
  <input type="text" placeholder="url" #url>

  <button (click)="addTutorial(name.value,url.value)">Add</button>
</div>
<div class="col-md-6"></div>
</div>
```

Create.component.ts

```
import { Observable } from 'rxjs/Observable';
import { Store } from '@ngrx/store';
import { Tutorial } from './../models/tutorial.model';
import * as TutorialActions from './../actions/tutorial.action';

@Component({
  selector: 'app-create',
  templateUrl: './create.component.html',
  styleUrls: ['./create.component.css']
})
export class CreateComponent implements OnInit {

  constructor(private store: Store<AppState>) { }
addTutorial(name, url) {
  this.store.dispatch(new TutorialActions.AddTutorial({name: name, url: url}));
}
  ngOnInit() {
  }

}
```

Read.component.html

```
<div class ="right" *ngIf="tutorials">
  <h3>Tutorials</h3>
  <ul>
    <li (click)="delTutorial(i)" *ngFor="let tutorial of tutorials | async; let i = index">
      <div class="col-md-12">{{tutorial.name}}</div>
      <div class="col-md-12"><a [href]="tutorial.url" target="_blank">
        {{tutorial.url}}
      </a></div>
    </li>
  </ul>
</div>
```

Read.component.ts

```
import { AppState } from './../app.state';

import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Store } from '@ngrx/store';
import { Tutorial } from './../models/tutorial.model';
import * as TutorialActions from './../actions/tutorial.action';

@Component({
  selector: 'app-read',
  templateUrl: './read.component.html',
  styleUrls: ['./read.component.css']
})
export class ReadComponent implements OnInit {

  tutorials: Observable<Tutorial[]>;
  constructor(private store: Store<AppState>) {
    this.tutorials = store.select('tutorial');
   }
delTutorial(index) {
  this.store.dispatch(new TutorialActions.RemoveTutorial(index));
}
  ngOnInit() {
  }

}
```

Sample